CISC 867: Deep Learning

Project #2

Supervisor:
Dr. Hazem Abbas

Done By:
Name : Esraa Ahmed Fouad Omar
ID : 20399123

Contents

List of Figures

1 Data

Fashion-MNIST consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels (see above), and represents the article of clothing as shown in figure [1]. Each training and test example is assigned to one of the following labels:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

Read Data

```
train_df = pd.read_csv('fashion-mnist_train.csv',sep=',')
test_df = pd.read_csv('fashion-mnist_test.csv', sep = ',')
print("Shape of train data" , train_df.shape )
print("Shape of test data" , test_df.shape)
train_df.head()
```

```
Shape of train data (60000, 785)
Shape of test data (10000, 785)
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	0	0	105	92	101	107	100
3	0	0	0	0	1	2	0	0	0	0	0	114	183	112	55	23	72
4	3	0	0	0	0	0	0	0	0	0	0	0	0	46	0	21	68

Figure 1: Sample of data

2 Explore Data

1. Check Type of our data : As you can see in figure[2], all of our features(784) are numerical. Also, the target column(label) is numerical.

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 359.3 MB
```

Figure 2: Type of our data

2. Check the Distribution of our data : As you can see in figure[3],a statistical summary of the input variables is provided showing that values of pixels are numeric and their range is approximately from 0 to 255.Also, values of label is numeric and their range is approximately from 0 to 9.

```
train_df.describe()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776
count	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	...	60000.000000	60000.000000
mean	4.500000	0.000900	0.006150	0.035333	0.101933	0.247967	0.411467	0.805767	2.198283	5.682000	...	34.625400	23.300683
std	2.872305	0.094689	0.271011	1.222324	2.452871	4.306912	5.836188	8.215169	14.093378	23.819481	...	57.545242	48.854427
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000
25%	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000
50%	4.500000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000
75%	7.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	58.000000	9.000000
max	9.000000	16.000000	36.000000	226.000000	164.000000	227.000000	230.000000	224.000000	255.000000	254.000000	...	255.000000	255.000000

8 rows × 785 columns

Figure 3: Distribution of our data using Describe function

Note that :

The figure [3] doesn't contain all the information about all columns. I couldn't take a snap shoot for all columns. The full image is in the notebook, you can see it

Let's visualize our data to see the distribution in more details.we can see from figure [4] that most pixels are in the range from 0 to 255. The rest is varying over a small range. the data points is far from each other. we will normalize data .Normalizing reduces the range of the input values over

which the backpropagation algorithm has to work.Normalization also prevents one input variable varying over a small range while another over a big range. This can create a minor headache for the backpropagation.A smaller and uniform range for all pixels increases the chance for the convergence.

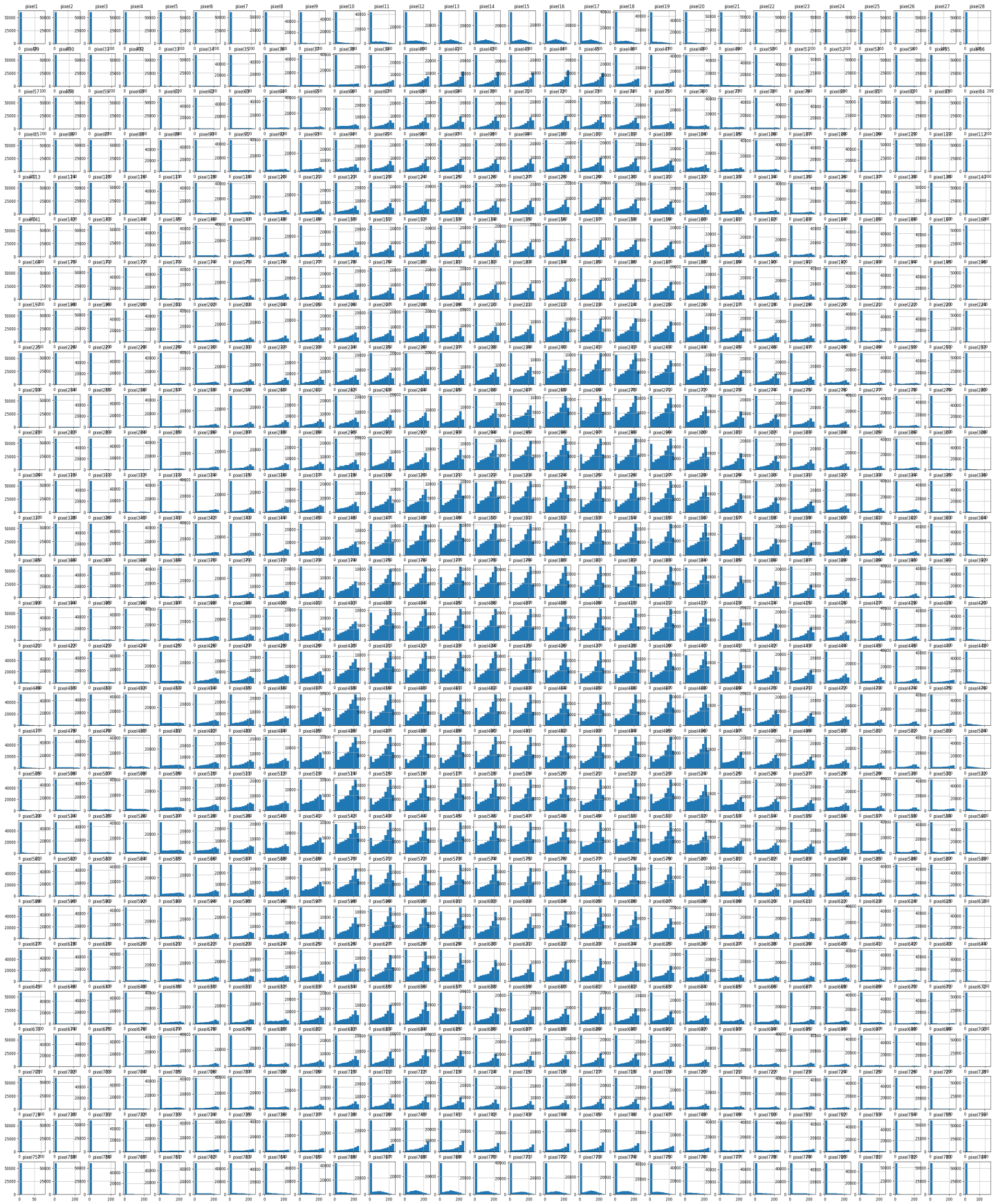


Figure 4: Distribution of our data using Visualization

3. Check Null values: As you can see in figure [5], there is no null values in our data

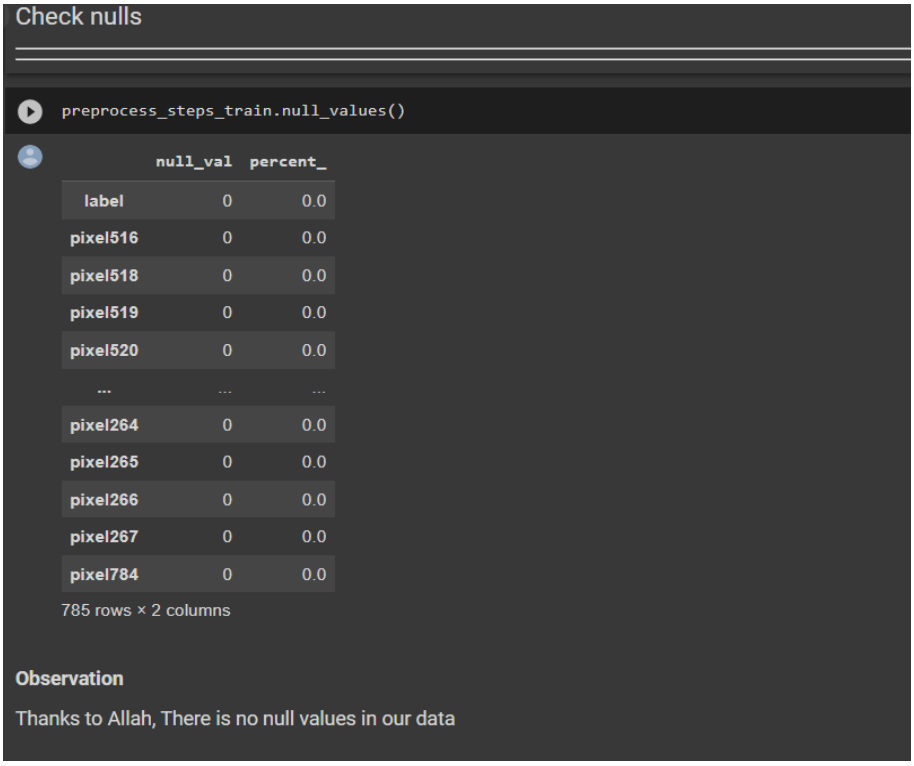


Figure 5: sample of

4. Check Duplication: As you can see in figure [6], there is about 43 duplicated rows in our data.We will drop them in next section

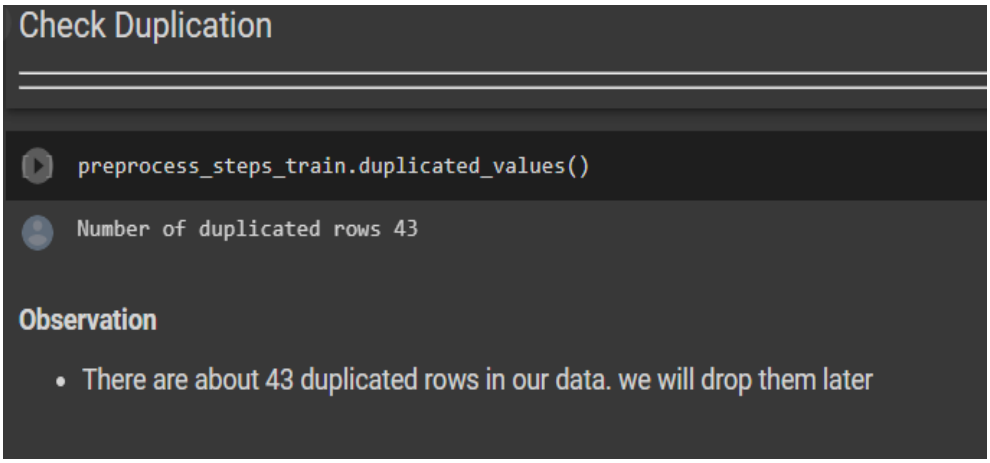


Figure 6: Number of duplicated rows in our data

5. Check Constant features : Here, we check if there are columns contain only one value a cross all samples. As you can see in figure [7], there is no constant features in our data

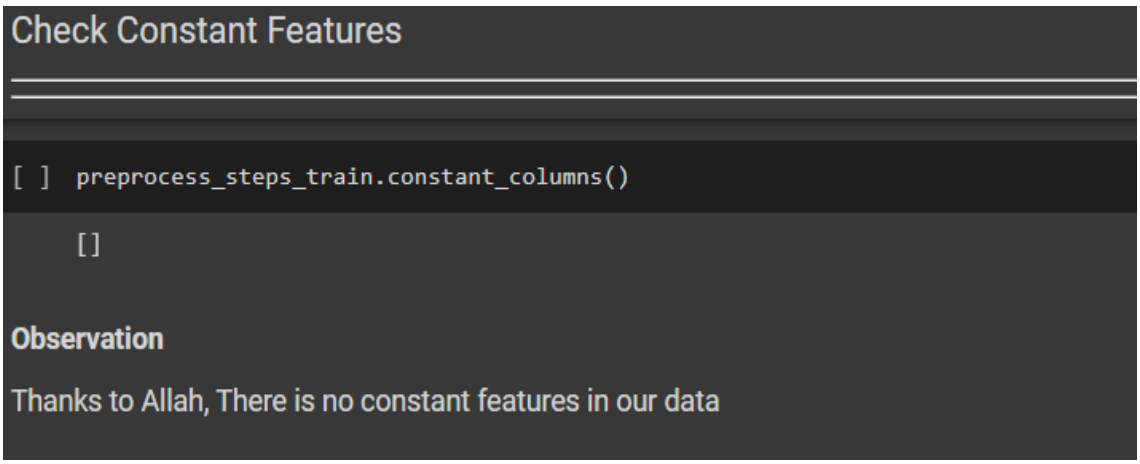
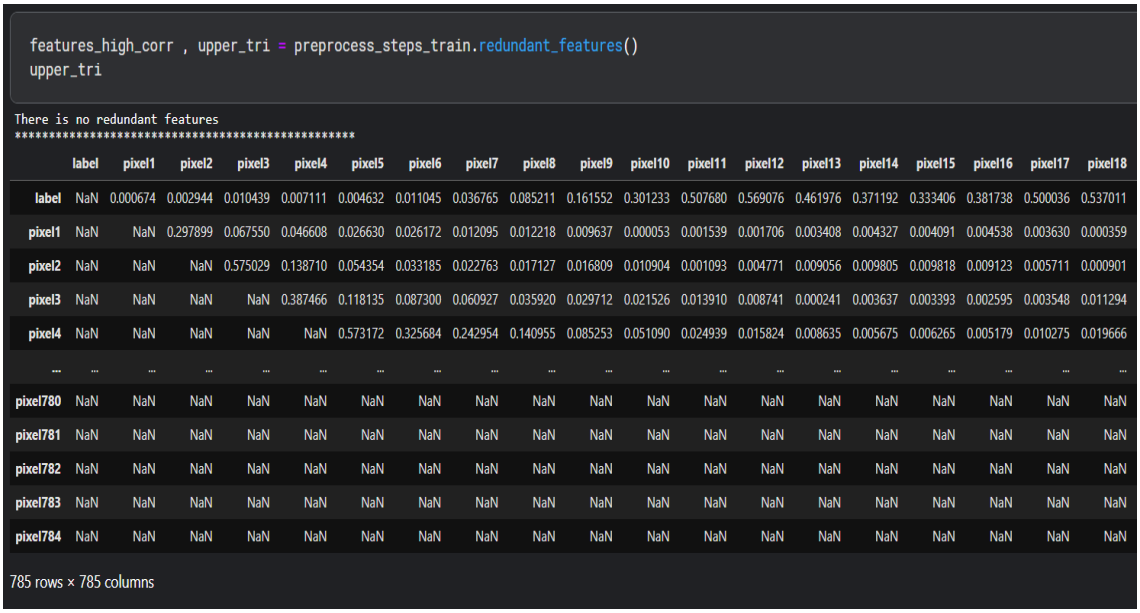


Figure 7: Number of constant features in our data

6. Check redundant features : Here, we check if there are columns are highly correlated (corr more than 0.98). As you can see in figure 8, there are no columns are highly correlated.



	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16	pixel17	pixel18
label	NaN	0.000674	0.002944	0.010439	0.007111	0.004632	0.011045	0.036765	0.085211	0.161552	0.301233	0.507680	0.569076	0.461976	0.371192	0.333406	0.381738	0.500036	0.537011
pixel1	NaN	NaN	0.297899	0.067550	0.046608	0.026630	0.026172	0.012095	0.012218	0.009637	0.000053	0.001539	0.001706	0.003408	0.004327	0.004091	0.004538	0.003630	0.000359
pixel2	NaN	NaN	NaN	0.575029	0.138710	0.054354	0.033185	0.022763	0.017127	0.016809	0.010904	0.001093	0.004771	0.009056	0.009805	0.009818	0.009123	0.005711	0.000901
pixel3	NaN	NaN	NaN	NaN	0.387466	0.118135	0.087300	0.060927	0.035920	0.029712	0.021526	0.013910	0.008741	0.000241	0.003637	0.003393	0.002595	0.003548	0.011294
pixel4	NaN	NaN	NaN	NaN	NaN	0.573172	0.325684	0.242954	0.140955	0.085253	0.051090	0.024939	0.015824	0.008635	0.005675	0.006265	0.005179	0.010275	0.019666
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
pixel780	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
pixel781	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
pixel782	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
pixel783	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
pixel784	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 8: Number of redundant features in our data

Also, we visualize the correlation using heapmap. Unfortunately, Due to that there are a huge number of columns , we can't plot them in one plot .So, We only plot the first 50 columns as shown in figure [9]

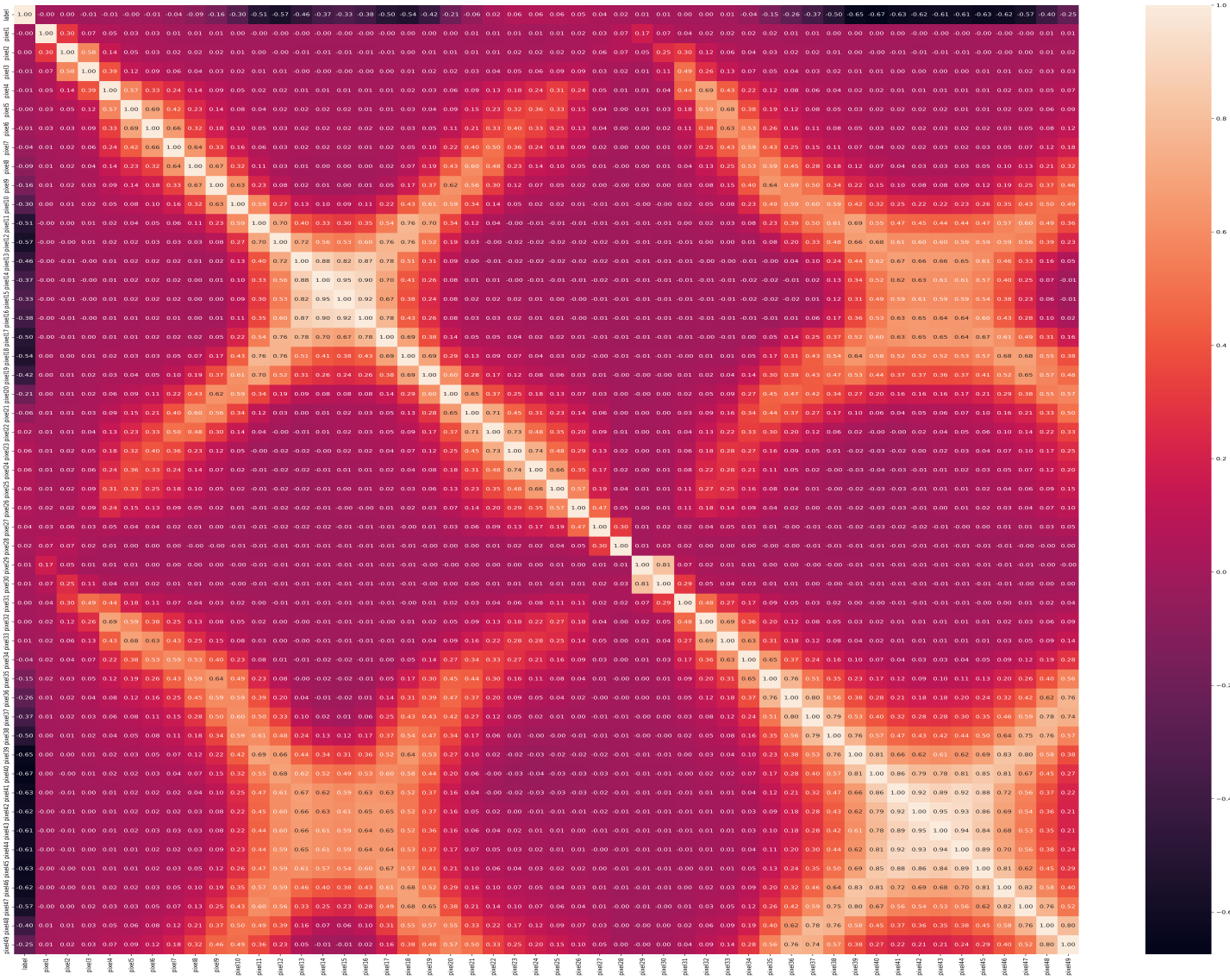


Figure 9: Visualize correlation using heapmap

7. Check cardinality: As you can see, There are no columns are highly cardinality as shown in Figure [10].

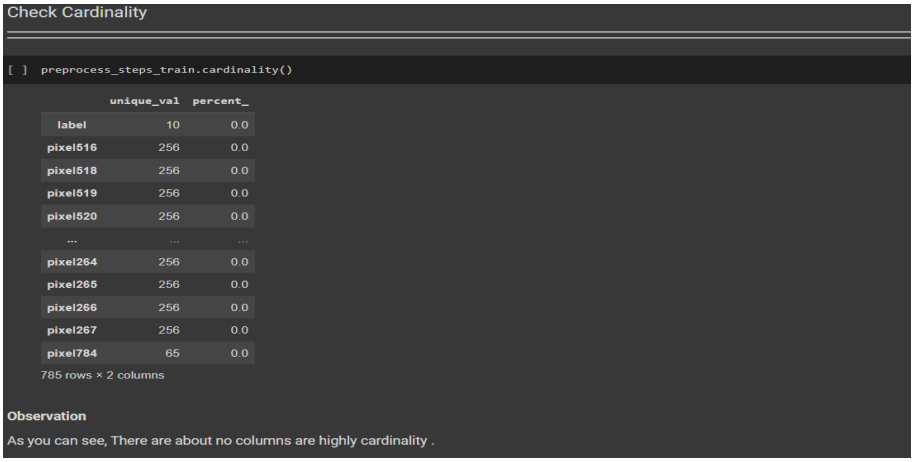


Figure 10: sample of columns and its unique values

8. Check Imbalance in classes: As you can see , we have 10 classes in label column.Our labels are balanced as shown in the figure [12].

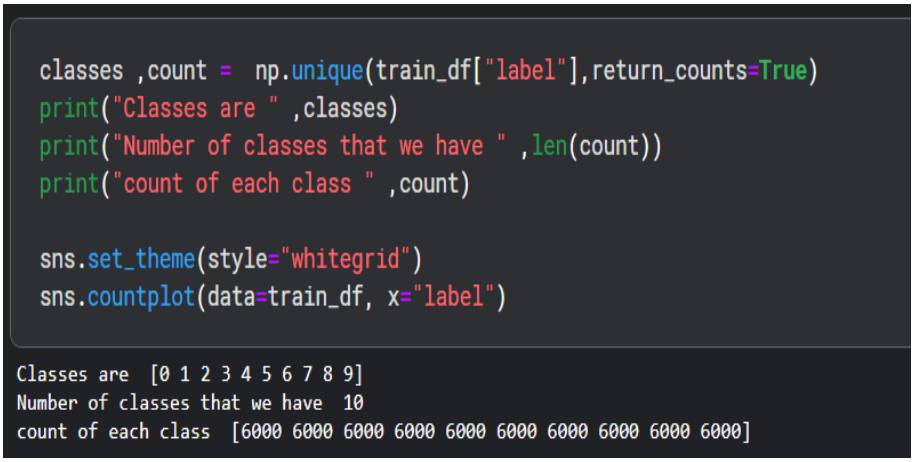


Figure 11: labels and their count

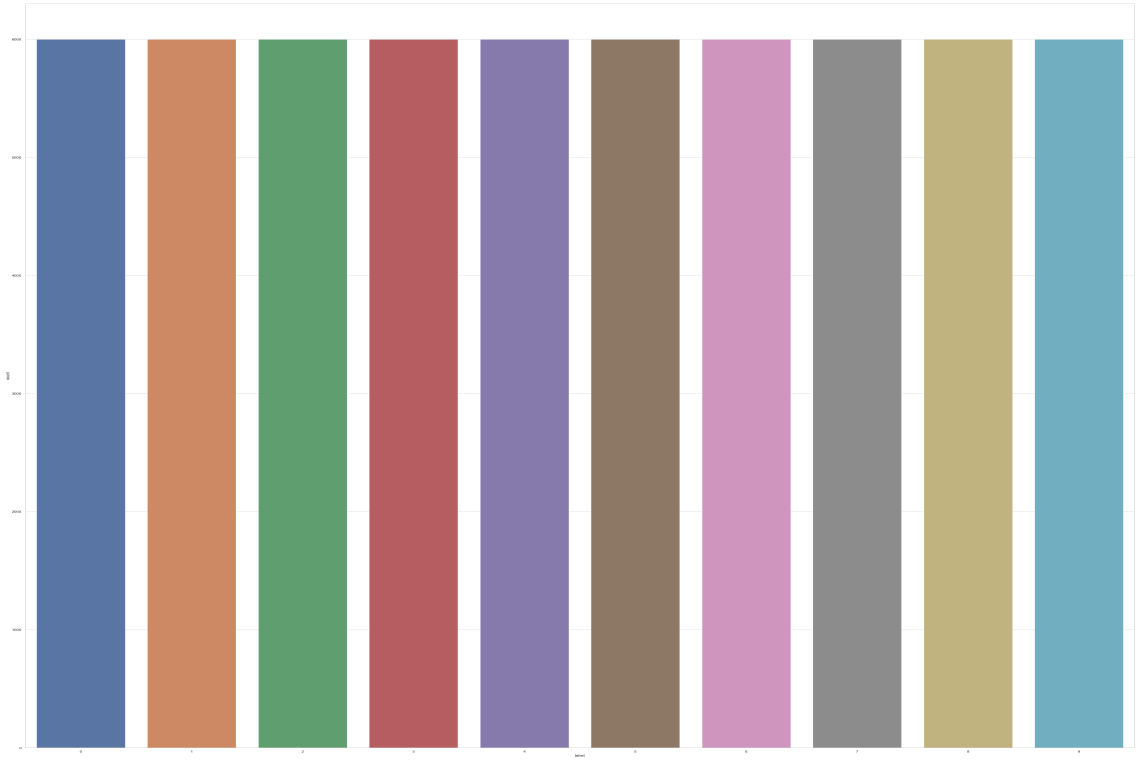


Figure 12: Count values in each class

3 Preprocessing Data

1. Drop Duplicated rows: From the previous section, we know that we need to drop duplicated rows. They are about 43 rows. As you can see, we successfully drop them as shown in figure [13].

```
train_df = preprocess_steps_train.drop_col()
preprocess_steps_train.duplicated_values()
```

```
Number of duplicated rows 0
```

Figure 13: Remove duplicated rows

2. we need to convert the dataframes into numpy arrays of float32 type which is the acceptable form for tensorflow and keras.

3. **Reshape Data into the acceptable form of Lenet-5 model :**

   (a) Convert data from columns form to image matrix 28×28 form then add padding to be 32×32 because it is the acceptable image size Lenet-5 model.

   (b) Normalize data by dividing them on 255

   (c) Add the channels dimention so it will become (..., 32, 32, 1) instead of (..., 32, 32) because it is the acceptable form for tensorflow .

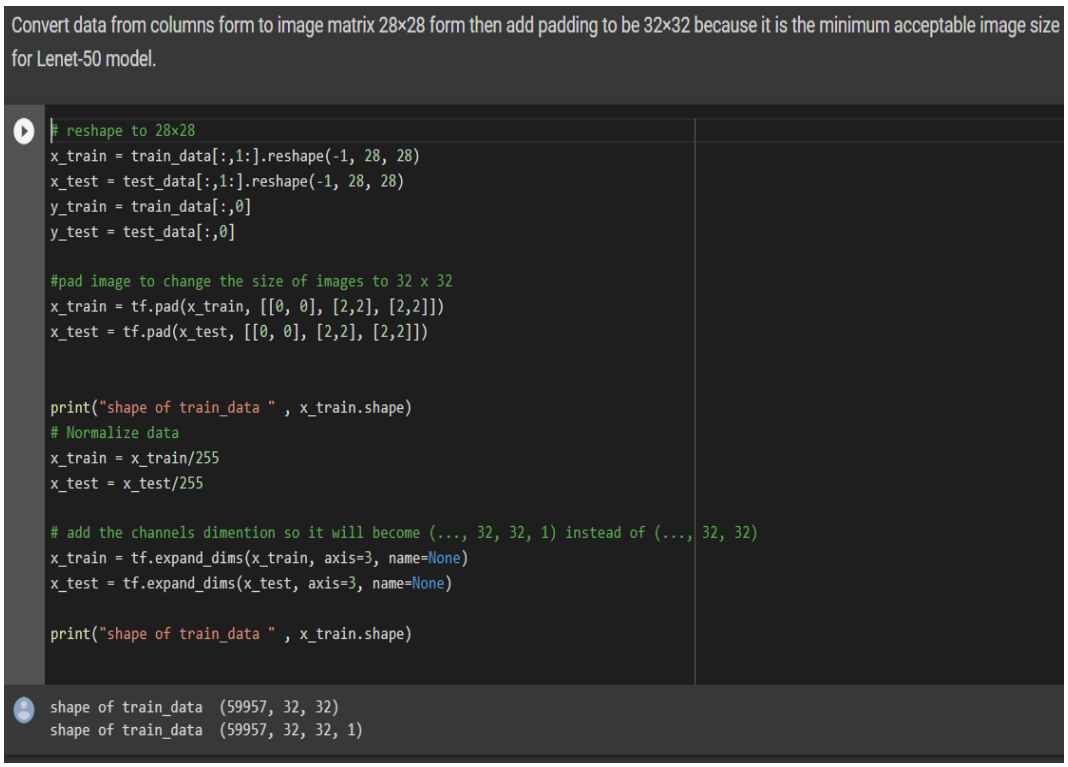


Figure 14: Shape of Input data for Lenet-5 model

4. **Reshape Data into acceptable form of pretrained models(VGG16 and Resnet152v2):**

   (a) Convert the images into 3 channels

   (b) Convert data from columns form to image matrix 28×28 form then Resize the images into 48 x 48 as required by VGG16 Resnet

   (c) Normalize data by dividing them on 255.

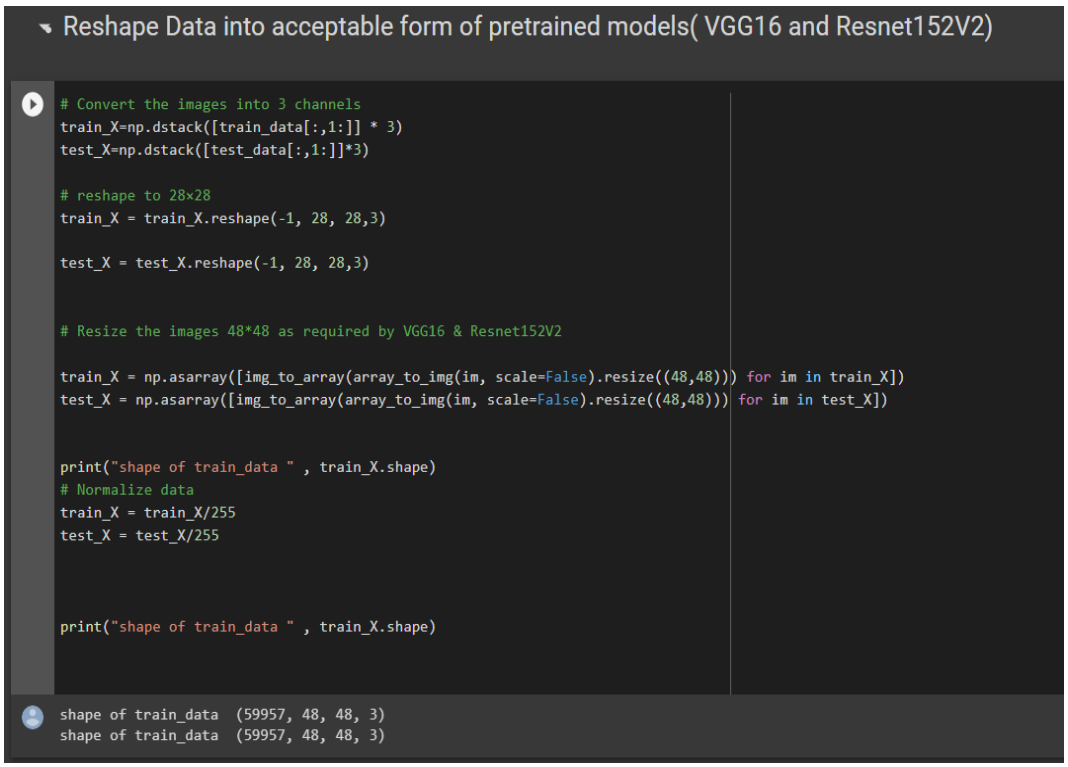


Figure 15: Shape of Input data for VGG16 and RESNET152V2 model

4 Encode Labels

As we Knew from the section of Explore Data, the classes is in categories form (from 0 to 9). In this section, we convert labels into one hot encoding as shown in figure [16].

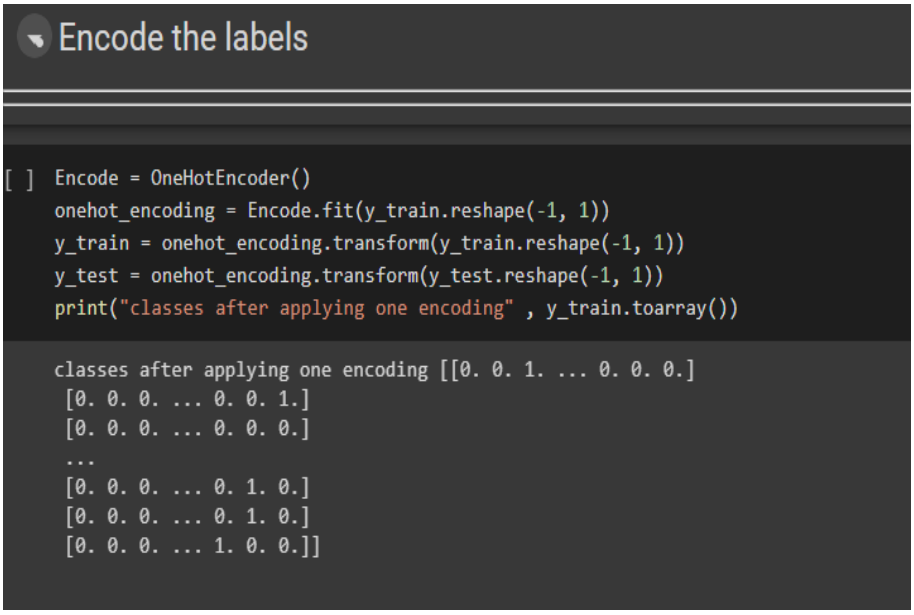


Figure 16: Label encode

5 Draw some of the images

Here, we display some images.we display the first 50 images in our data as shown in figure [18]

Visualize - Draw Images

```
[ ] ROW_IMG = 10
    N_ROWS = 5
    fig = plt.figure()
    plt.figure(figsize=(13,15))
    labels = ['T_shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
    for index in range(1, ROW_IMG * N_ROWS + 1):
        plt.subplot(N_ROWS, ROW_IMG, index)
        plt.axis('off')
        plt.imshow(x_train[index], cmap='binary')
        plt.title(labels[int(y_train[index])])
```

Figure 17: Code to display images

Figure 18: sample of MNIST fashion datasets

6 Split Data

The data is already split into train and test as we mentioned before. In fitting the model, we split train data into train and validation sets with ratio 80 to 20 respectively.

7 Build Model

In this Section, we implemented functions to built our models.

1. **LeNet-5 Model:** we implemented a LeNet-5 network to recognize the Fashion MNIST digits. The input to this model is a 32 X 32 grayscale image hence the number of channels is one.So, we convert our data into this shape to be compitable.

- We then apply the first convolution operation with the filter size 5X5 and we have 6 such filters. As a result, we get a feature map of size 28X28X6.
- After the first pooling operation, we apply the average pooling and the size of the feature map is reduced by half.
- Next, we have a convolution layer with sixteen filters of size 5X5. Again the feature map changed it is 10X10X16. After this, we again applied an average pooling or subsampling layer, which again reduce the size of the feature map by half i.e 5X5X16.
- Then we have a final convolution layer of size 5X5 with 120 filters. Leaving the feature map size 1X1X120. After which flatten result is 120 values.
- After these convolution layers, we have a fully connected layer with eighty-four neurons.
- At last, we have an output layer with ten neurons since the data have ten classes.

we used adam as optimizer and categorical cross entropy as loss Function.

```
def build(self, hp):
  model = models.Sequential()
  model.add(layers.Conv2D(6, 5, activation='tanh', input_shape=x_train.shape[1:]))
  model.add(layers.AveragePooling2D(2))
  model.add(layers.Activation('sigmoid'))
  model.add(layers.Conv2D(16, 5, activation='tanh'))
  model.add(layers.AveragePooling2D(2))
  model.add(layers.Activation('sigmoid'))
  model.add(layers.Conv2D(120, 5, activation='tanh'))
  model.add(layers.Flatten())
  model.add(layers.Dense(84, activation='tanh'))
  model.add(layers.Dense(10, activation='softmax'))
  model.summary()


  # Tune learning rate for Adam optimizer with values from 0.01, 0.001, or 0.0001
  hp_learning_rate = hp.Choice("learning_rate", values=[1.0 ,1e-2, 1e-3, 1e-4])

  # Define optimizer, loss, and metrics
  model.compile(optimizer= tf.keras.optimizers.Adam(learning_rate=hp_learning_rate),
                loss=tf.keras.losses.CategoricalCrossentropy(),
                metrics=["accuracy"])

  return model
```

Figure 19: Function to build LeNet-5 model

2. **VGG-16 Model:**

   (a) We setup input layer

   (b) We removed the following top (last) layer from our customized layer :

      i. block5_pool (MaxPooling2D) (None, 7, 7, 512)
      ii. flatten (Flatten) (None, 25088)
      iii. fc1 (Dense) (None, 4096)
      iv. fc2 (Dense) (None, 4096)
      v. predictions (Dense) (None, 1000)

   (c) access the current last layer of the model and add flatten and 3 dense layers. Each one is followed by dropout layer after it. The dropout layer prevents overfit

   (d) Add dense layer to the final output layer

```
def preTrainedVgg16(input_layer):
    preTrainedModelVgg16 = VGG16(input_tensor = input_layer, include_top = False, weights='imagenet')
    VGG16layers=preTrainedModelVgg16.layers
    print("Number of layer Vgg16 : ",len(VGG16layers))
    for layer  in range(len(VGG16layers)-5):
           VGG16layers[layer].trainable = False

    preTrainedModelVgg16.summary()
    #Vgg16 Model
    x=Flatten()(preTrainedModelVgg16.output)
    #Fully Connection Layer
    # FC1
    x=Dense(1024, activation="relu")(x)
    # FC2
    x=Dense(1024, activation="relu")(x)
    # FC3
    x=Dense(1024, activation="relu")(x)
    #Dropout to avoid overfitting effect
    x=Dropout(0.5)(x)
    # FC4
    x=Dense(512, activation="relu")(x)
    # FC5
    x=Dense(512, activation="relu")(x)
    #Dropout to avoid overfitting effect
    x=Dropout(0.4)(x)
    # FC6
    x=Dense(256, activation="relu")(x)
    # FC7
    x=Dense(64, activation="relu")(x)
    # FC8
    x=Dense(64, activation="relu")(x)
    #Dropout to avoid overfitting effect
    x=Dropout(0.2)(x)
    #output layer
    x=Dense(10,activation="softmax")(x)

    modelpreTrainedModelVgg16=models.Model(inputs=input_layer,outputs=x)
    return modelpreTrainedModelVgg16
```

Figure 20: Function to build VGG-16 model

3. **Resnet152v2 Model:**

   (a) We setup input layer

   (b) We removed the top (last) layer from our customized layer

   (c) access the current last layer of the model and add flatten and 2 dense layers. Each one is followed by dropout layer after it. The dropout layer prevents overfit

   (d) Add dense layer to the final output layer

```
def preTrainedResNet152V2 (input_layer):
    preTrainedModelResNet152V2  = ResNet152V2 (input_tensor = input_layer, include_top = False, weights="imagenet")
    ResNet152V2layers=preTrainedModelResNet152V2.layers
    print("Number of layer ResNet152V2 : ",len(ResNet152V2layers))
    for layer  in range(len(ResNet152V2layers)-64):
        ResNet152V2layers[layer].trainable = False

    preTrainedModelResNet152V2.summary()
    #ResNet152V2 Model
    x=Flatten()(preTrainedModelResNet152V2.output)

    #Fully Connection Layers
    # FC1
    x=Dense(1024, activation="relu")(x)
    # FC2
    x=Dense(1024, activation="relu")(x)
    # FC3
    x=Dense(1024, activation="relu")(x)
    # FC4
    x=Dense(1024, activation="relu")(x)
    # #Dropout to avoid overfitting effect
    x=Dropout(0.2)(x)
    # FC5
    x=Dense(512, activation="relu")(x)
    # FC6
    x=Dense(512, activation="relu")(x)
    # FC7
    x=Dense(256, activation="relu")(x)
    # FC8
    x=Dense(256, activation="relu")(x)
    # #Dropout to avoid overfitting effect
    x=Dropout(0.2)(x)
    #output layer
    x=Dense(10,activation="softmax")(x)
    #concatenation layers
    modelResNet152V2=models.Model(inputs=input_layer,outputs=x)
    modelResNet152V2.summary()
    return modelResNet152V2
```

Figure 21: Function to build RESENT-152V12 model

8 Tune Hyper parameters

In our project, we tried 3 different hyper parameters (batch size , learning rate and epochs) to get the best hyper parameters that achieve the best performance on LeNet-5. we used random search in keras tuner

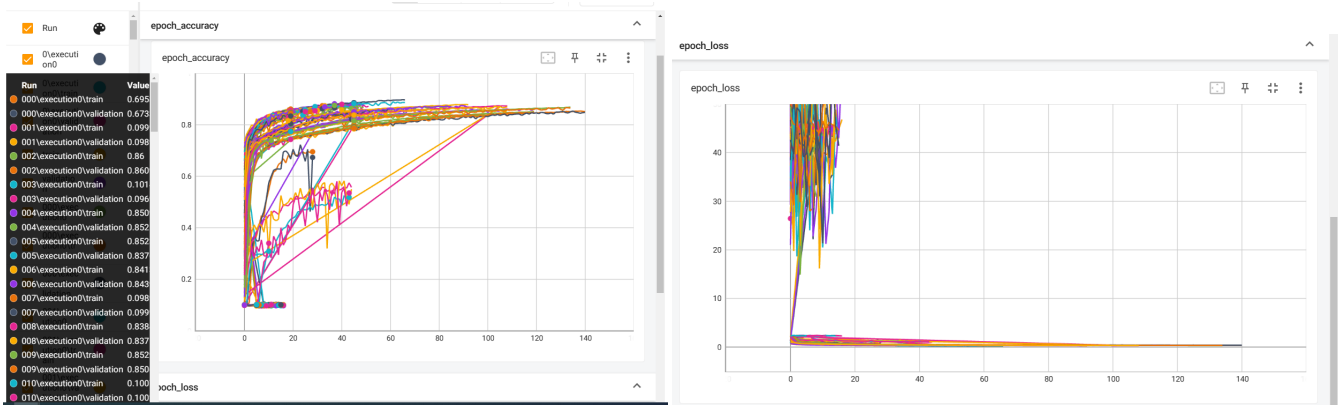
to create different combination between these hyper parameters. This function performs 100 trails with different combination in our search space.

```python
def tune_model(no_trails , xtrain, ytrain):
    tuner = kt.RandomSearch(
        MyHyperModel(),
        objective="val_accuracy",
        max_trials=no_trails,
        overwrite=True,
        directory="my_dir",
        project_name="tune_hypermodel",)
    # Run the hyperparameter search. The arguments for the search method are the same as those used for tf.keras.model.fit in addition to the callback above.

    tuner.search(xtrain, ytrain.toarray(),  validation_split=0.2, callbacks=[stop_early ,csv_logger , tf.keras.callbacks.TensorBoard("/tmp/tb_logs")] )
    # Get the optimal hyperparameters
    best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

    print(f"""
    The hyperparameter search is complete. The optimal number of units in the first densely-connected
    layer is and the optimal learning rate for the optimizer
    is {best_hps.get('learning_rate')}.
     """)
    return tuner , best_hps
```

Figure 22: funtion for tuning hyperparameters

After 100 trails, this function returns the best hyperparameters. Then we use these hyperparameters to fit the model.

we use tensorboard to track the convergence curve for LeNet-5 in each trail. The figure below reflect the convergence curve for LeNet-5 in each trail.



(a) Accuracy curve

(b) Loss curve

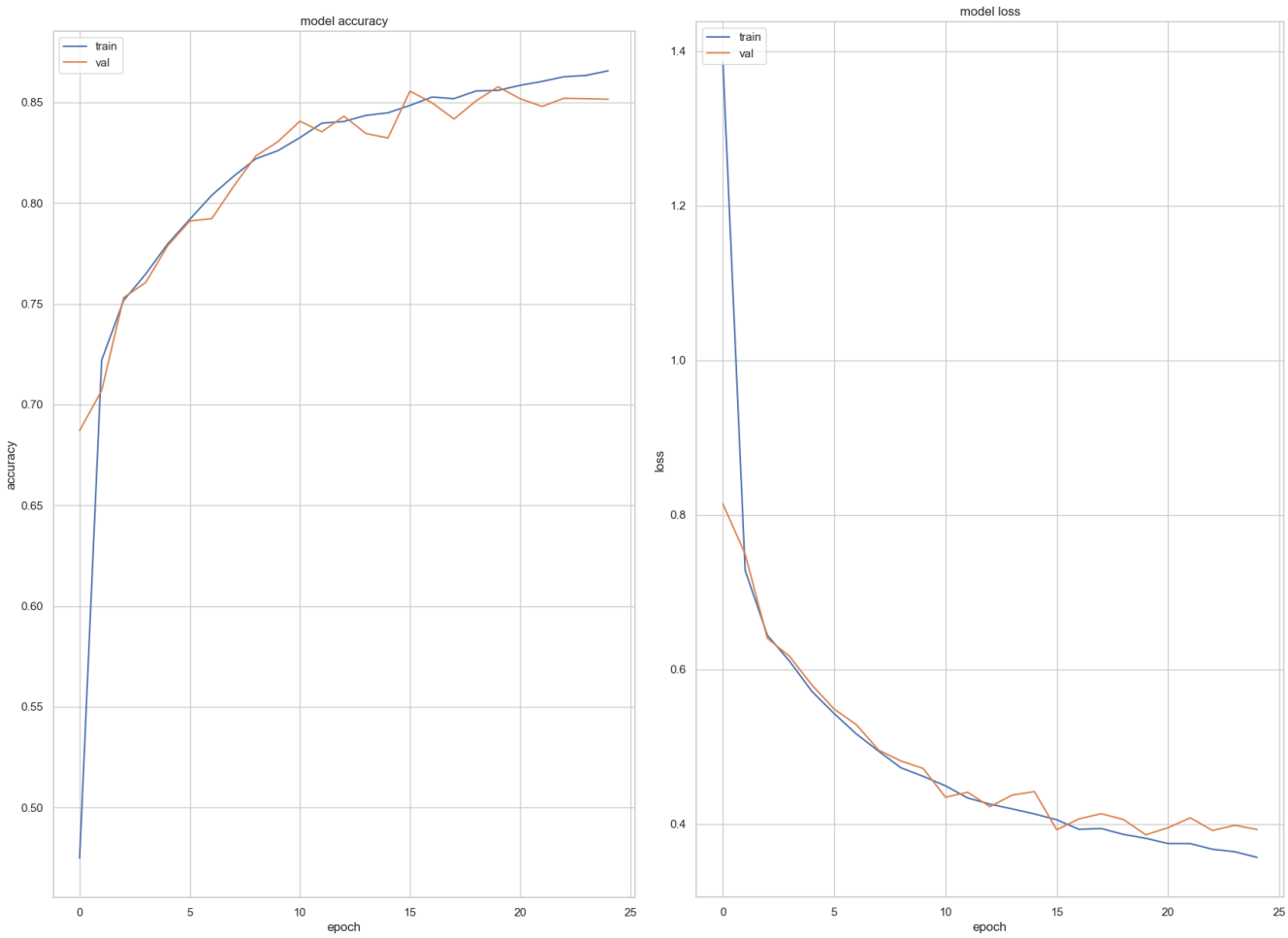
Figure 23: plots of trail curves

we pass the best hyperparameters that we got to fit_best_model function to train the LeNet-5 on these hyperparameters.

```python
def train_best_model(best_hyp ,xtrain , ytrain , xtest ,ytest , cross_validation = False ):
    model = tuner.hypermodel.build(best_hyp)
    epochs_ = best_hps.get('epochs')
    batch_size_ = best_hps.get('batch_size')
    if cross_validation :
        # Create StratifiedKFold object.
        kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
        lst_accu_stratified = []
        history_fold = []
        for train_index, test_index in kfold.split(xtrain , np.argmax(ytrain.toarray() , axis = 1)):
            print("Start Fold.....")
            x_train_fold, x_test_fold = tf.gather_nd(xtrain,indices=train_index.reshape(-1,1)), tf.gather_nd(xtrain,indices=test_index.reshape(-1,1))
            y_train_fold, y_test_fold = ytrain[train_index], ytrain[test_index]
            history = model.fit(x_train_fold,y_train_fold.toarray(),epochs = epochs_ , batch_size = batch_size_ ,
                                validation_data=(x_test_fold ,y_test_fold.toarray()), callbacks=[stop_early ,csv_logger , tf.keras.callbacks.TensorBoard("/tmp/tb_logs")])
            accuracy , report = evaluate_model(xtest, ytest , model)
            lst_accu_stratified.append(accuracy)
            history_fold.append(history)
            print("End Fold.....")

        # Print the output.
        print('List of possible accuracy:', lst_accu_stratified)
        print('\nMaximum Accuracy That can be obtained from this model is:',
              max(lst_accu_stratified)*100, '%')
        print('\nMinimum Accuracy:',
              min(lst_accu_stratified)*100, '%')
        print('\nOverall Accuracy:',
            statistics.mean(lst_accu_stratified)*100, '%')


        return  statistics.mean(lst_accu_stratified) ,history_fold

    else:

        history = model.fit(xtrain, ytrain.toarray(), epochs = epochs_ , batch_size = batch_size_, validation_split=0.2,verbose=5, callbacks=[stop_early ,csv_logger , tf.keras.callbacks.TensorBoard("/tmp/tb_logs")])

        return model , history
```

Figure 24: Fit best model Function

1. **LeNet-5 Without Cross Validation**:
   we passed the best hyperparameters that we got to fit_best_model function. After fitting the model with best hyperparamters, we got the pretty training curve. The curve reflect the model is good. There is no overfitting. Although the accuracy isn't large enough, it is only 88% an train data and 87% on test data, it can be improve by choosing another search space. we aren't lucky enough in choosing search space.



(a) LeNet-5 Accuracy

(b) LeNet-5 Loss

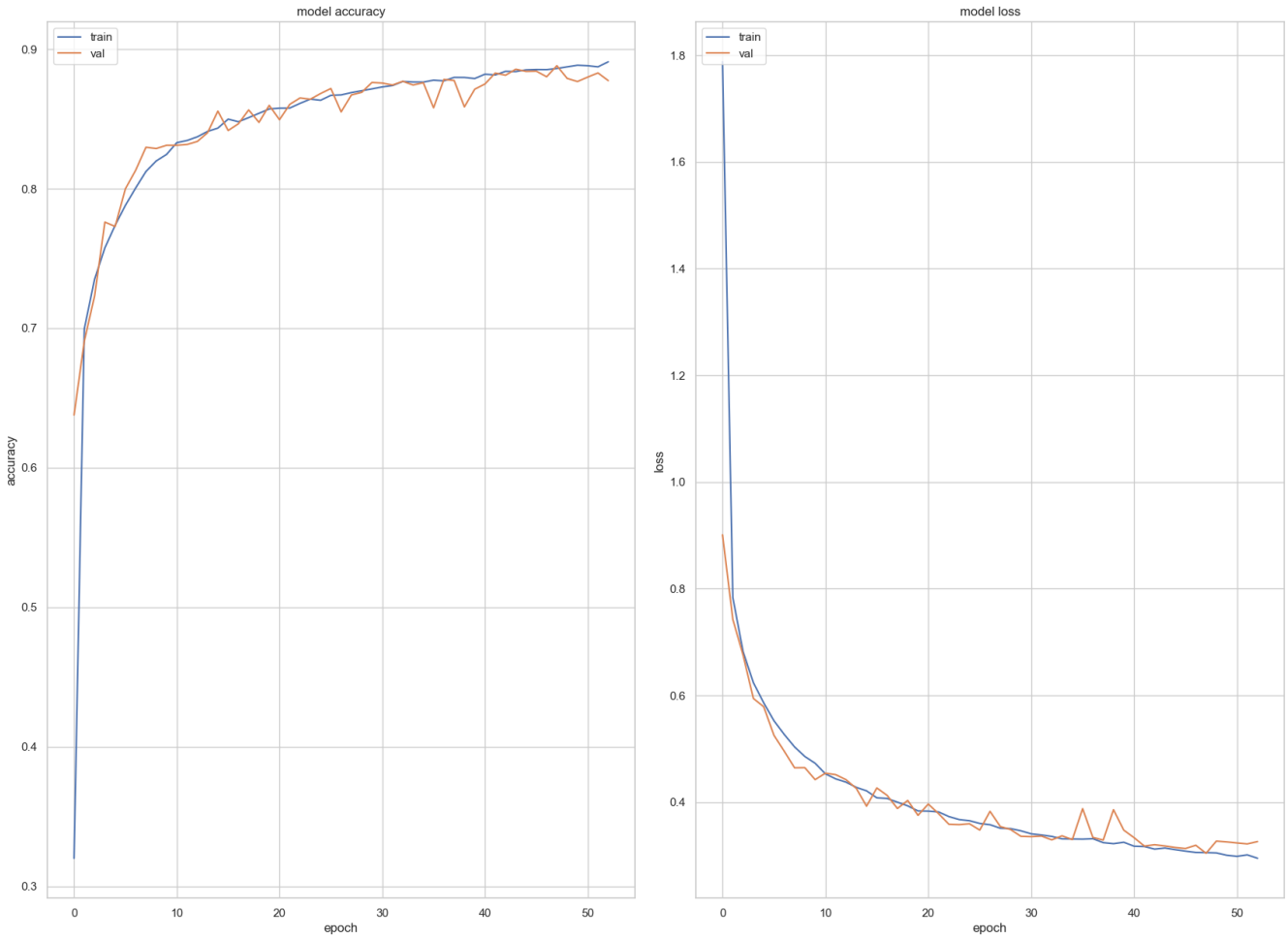
Figure 25: plots of LeNet-5

we evaluated our model in test data and got 85.44% accuracy same as validation accuracy as shown in the figure below

```
classification report
              precision    recall  f1-score   support

           0       0.85      0.69      0.76      1000
           1       0.97      0.96      0.97      1000
           2       0.82      0.67      0.74      1000
           3       0.80      0.91      0.85      1000
           4       0.69      0.89      0.78      1000
           5       0.95      0.95      0.95      1000
           6       0.67      0.64      0.66      1000
           7       0.96      0.89      0.92      1000
           8       0.97      0.96      0.96      1000
           9       0.90      0.98      0.94      1000

    accuracy                           0.85     10000
   macro avg       0.86      0.85      0.85     10000
weighted avg       0.86      0.85      0.85     10000


test accuracy 0.8544
```

Figure 26: Evaluation Matrix of Lenet-5 with best hyper parameters

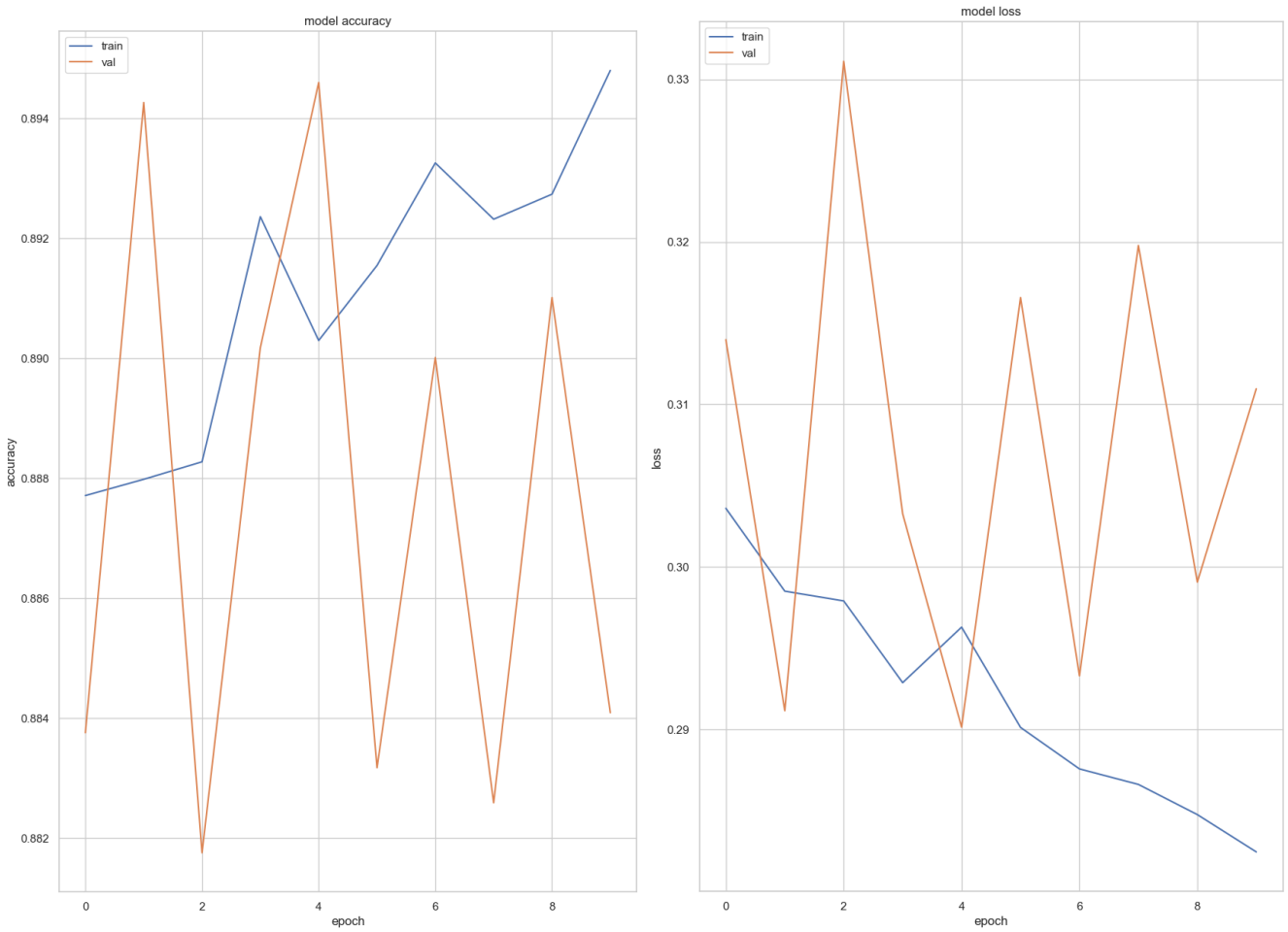
2. **LeNet-5 with Cross Validation :**
   Same as above, we pass best hyperparameters that we got to fit_best_model function but this time we pass cross_validation = True. After fitting the model with best hyperparameters, we got the below training curves.As you can see the curves is good. you may think that the curves isn't good as the image shows the difference is so large. But if you see the scaled in the axes, you will recognize that the curves is close to each other and the difference isn't significant.There is no overfitting in our model.



(a) Fold 1 accuracy

(b) Fold 1 loss

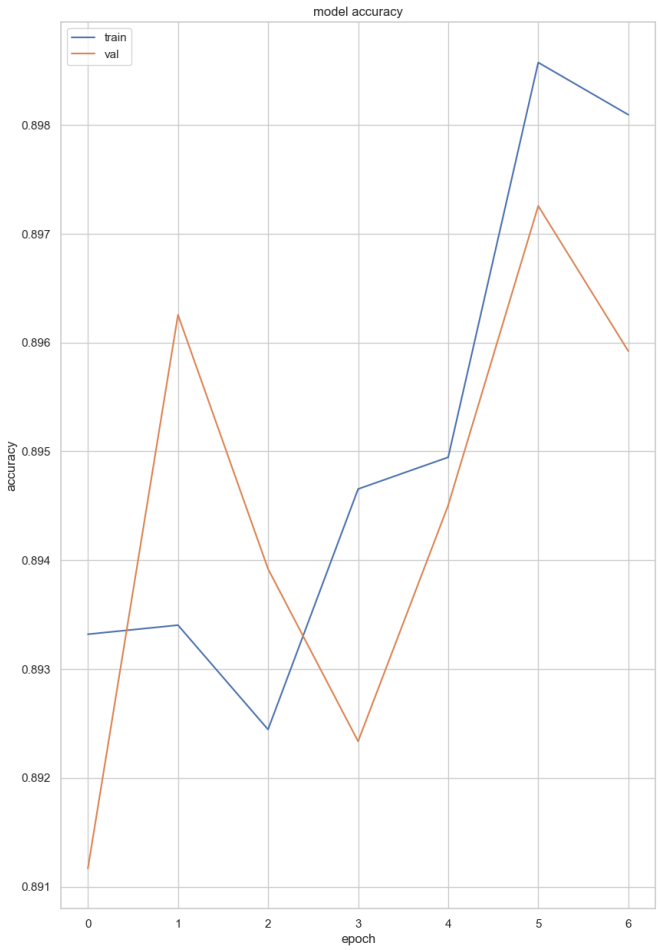
Figure 27: plots of fold 1



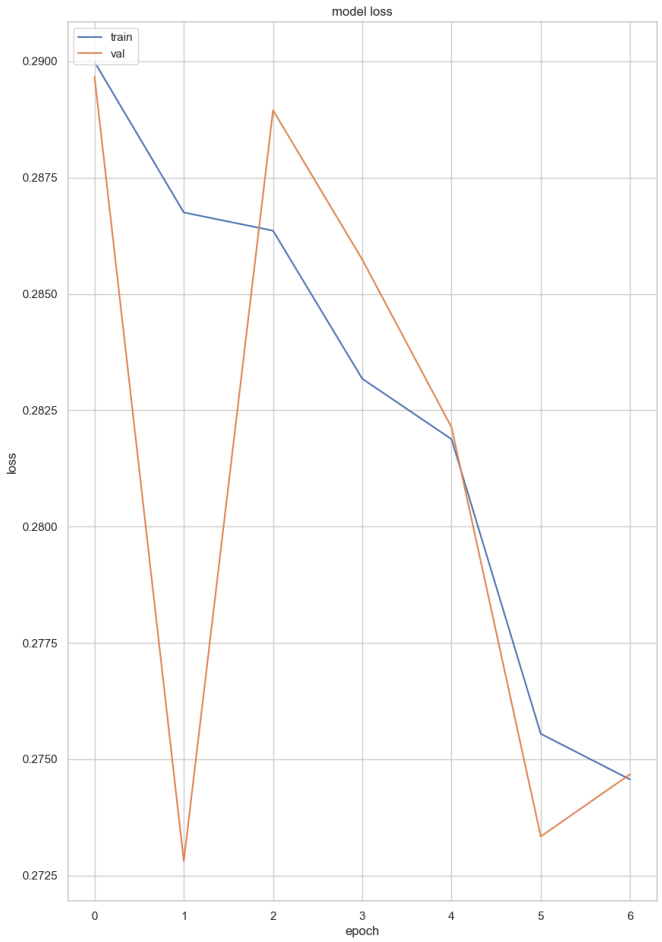
(a) Fold 2 accuracy

(b) Fold 2 loss

Figure 28: plots of fold 2

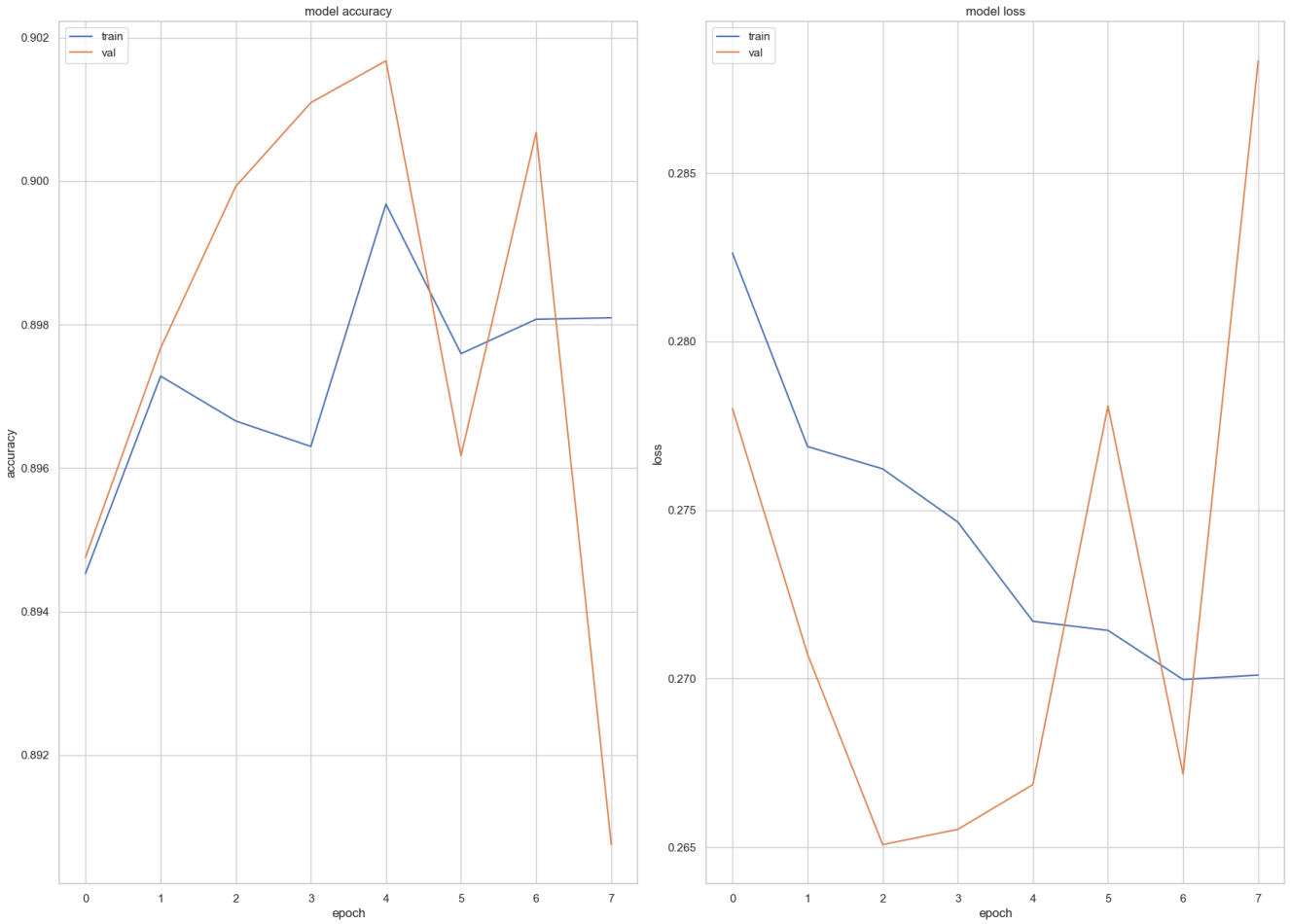


(a) Fold 3 accuracy



(b) Fold 3 loss

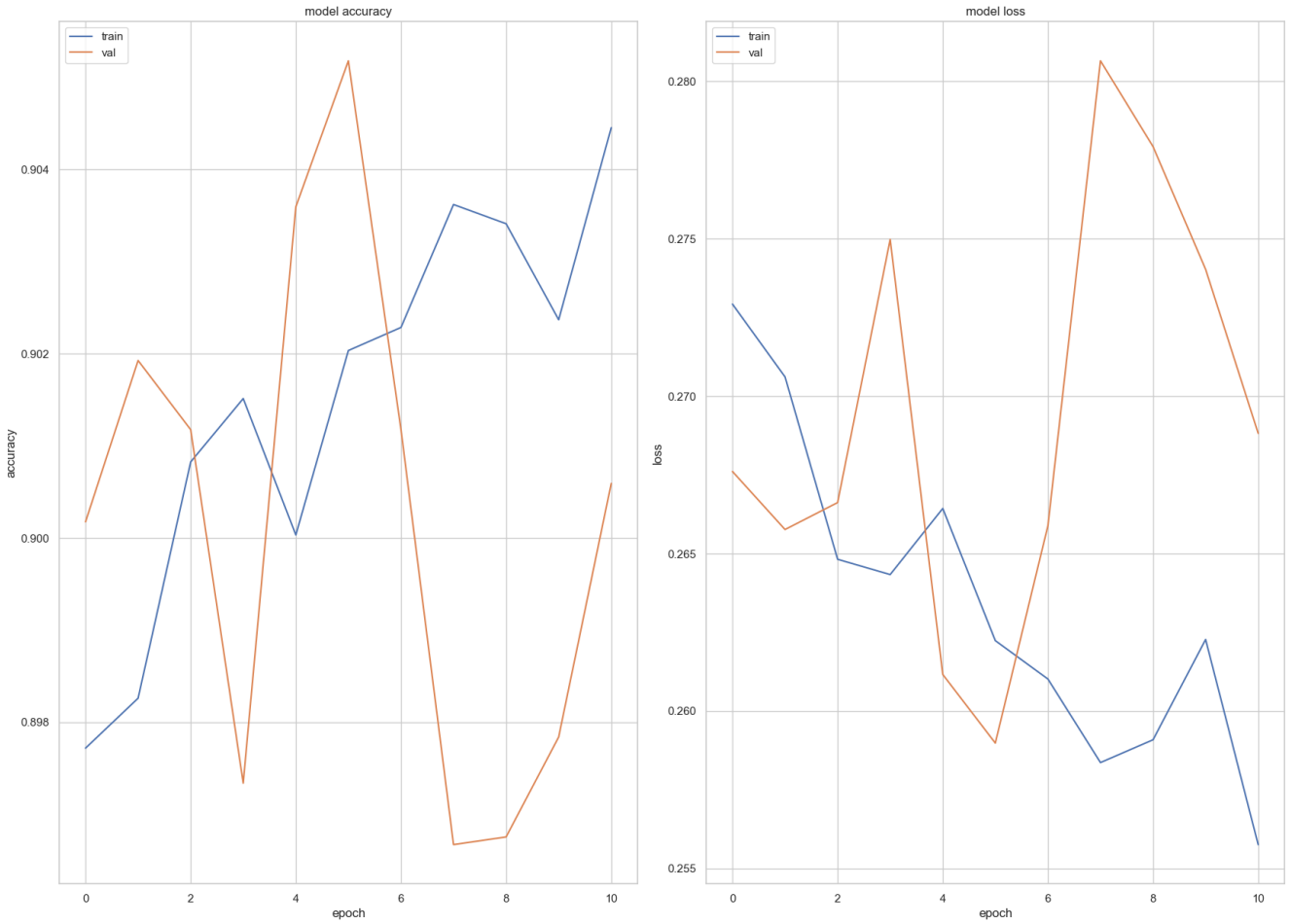
Figure 29: plots of fold 3



(a) Fold 4 accuracy

(b) Fold 4 loss

Figure 30: plots of fold 4



(a) Fold 5 accuracy

(b) Fold 5 loss

Figure 31: plots of Fold 5

The model with cross validation achieved (accuracy 88.87%) on test data which is higher than model without cross validation (88%)

```
List of possible accuracy: [0.8763, 0.8795, 0.8869, 0.8826, 0.8887]

Maximum Accuracy That can be obtained from this model is: 88.87 %

Minimum Accuracy: 87.63 %

Overall Accuracy: 88.28 %
```

Figure 32: Evaluation Matrix of Lenet-5 with best hyper parameters with cross validation

9 Transfer Learning

we choose VGG-16 and ResNet152V2 as we found them achieved high accuracy in keras application.we implemented function to fit these pretrained model after customizing them in the previous section(Build model).

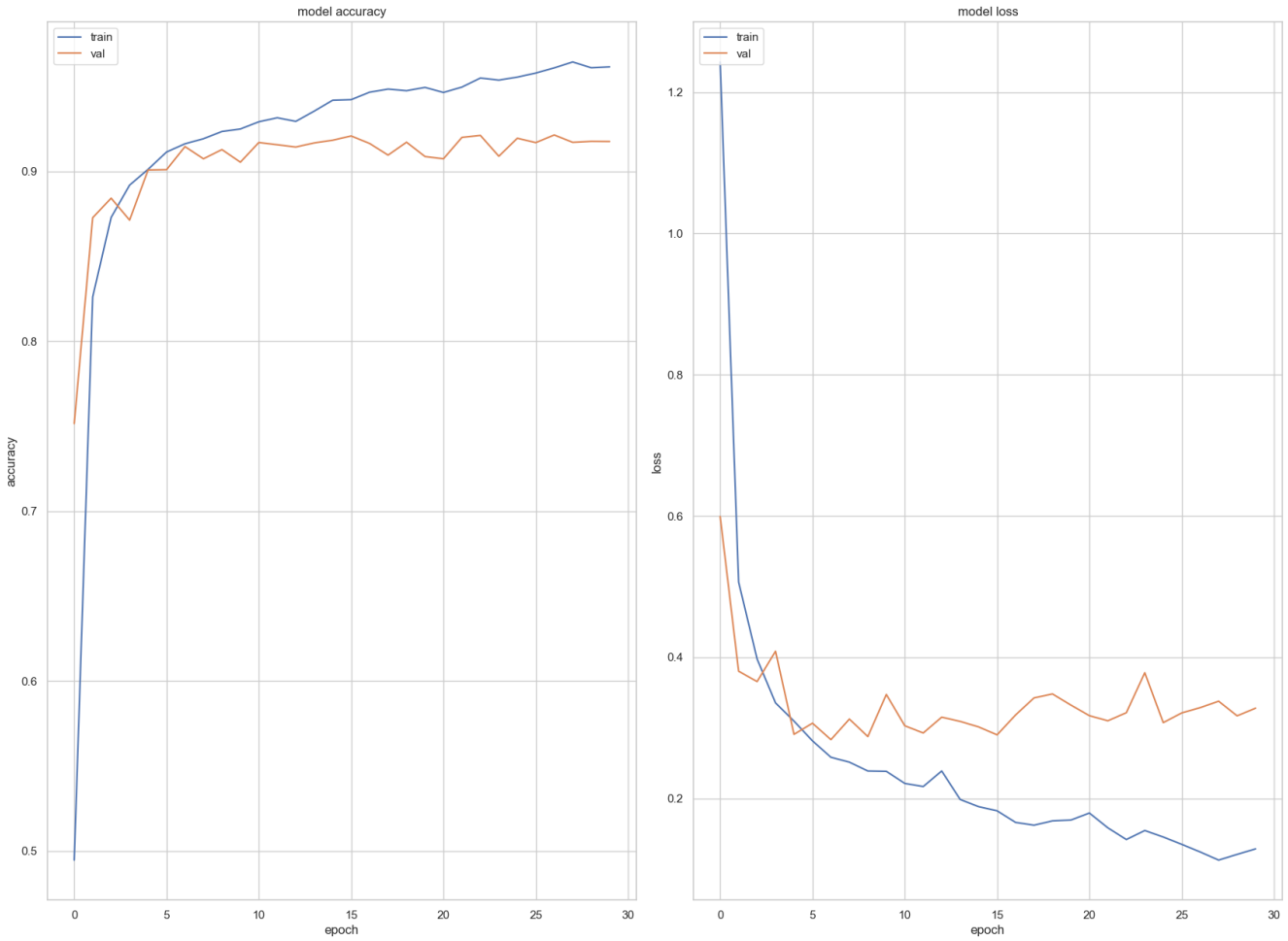
```
@jit(target_backend='cuda')
def fit_pretrained_model(xtrain , ytrain , model , epoch):
    model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
    history = model.fit(xtrain,ytrain , epochs=epoch, batch_size=128,  validation_split=0.2)
    return history , model
```

Figure 33: Function to fit pretrained model

we call this function and pass the model we want to train.

1. **Train VGG-16**:
   We pass the custimazed VGG-16 pretrained model we got from the previous section to fit_pretrained_model. This function will fit the custimazed VGG-16 pretrained model and returns history plot it

   After fitting our model, we got the following curves. The Curves is pretty good. There is no significant overfitting. The accuracy in train data is 96.19% and accuracy in validation data is 91.7 %. So, there is no significant difference between them. As you see there is a significant improvement in acccuracy than LeNet-5.



(a) VGG16 Accuracy

(b) VGG16 loss

Figure 34: plots of VGG16

we evaluated our model in test data and got 92.37% accuracy same as validation accuracy as shown in the figure below

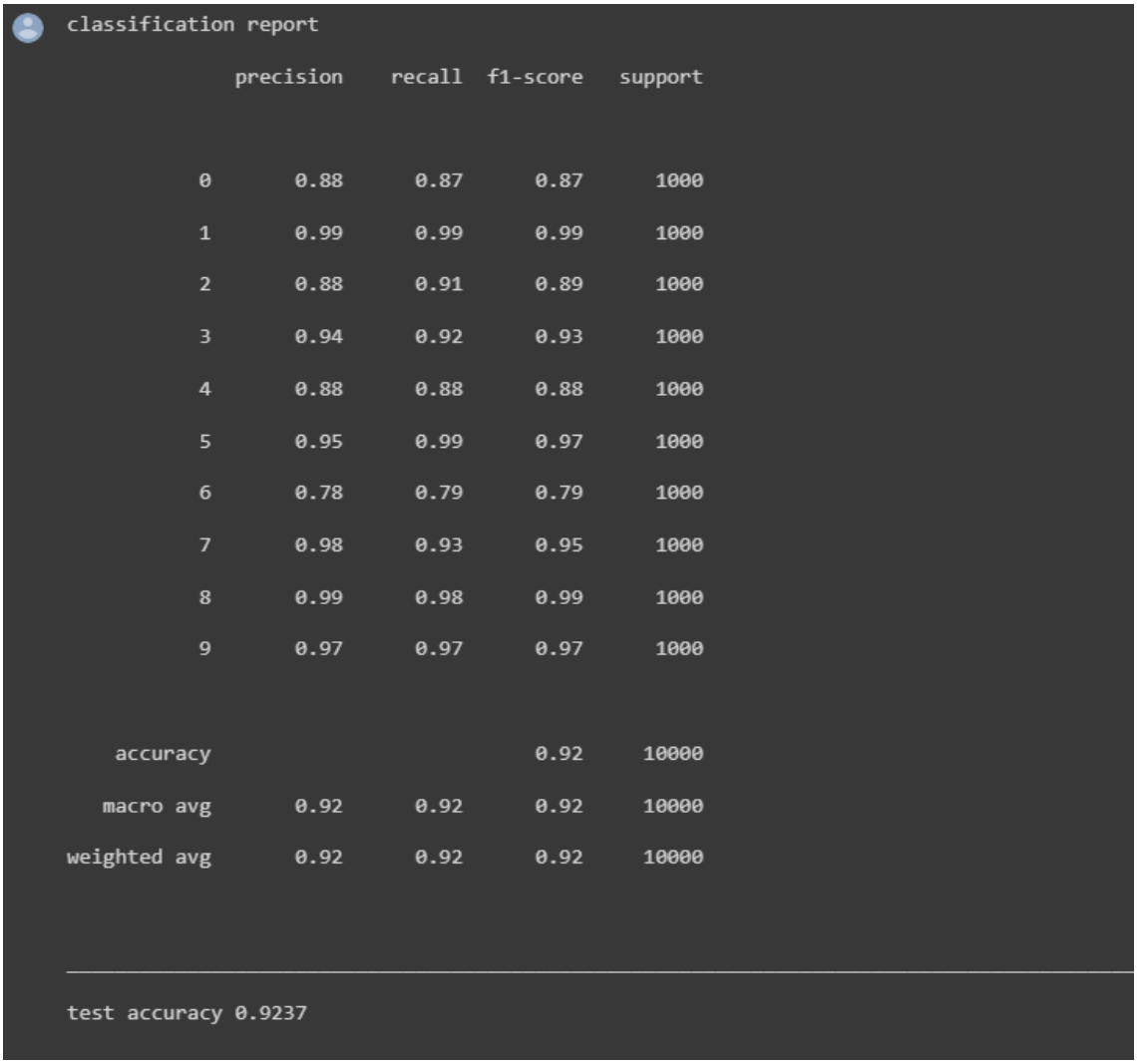
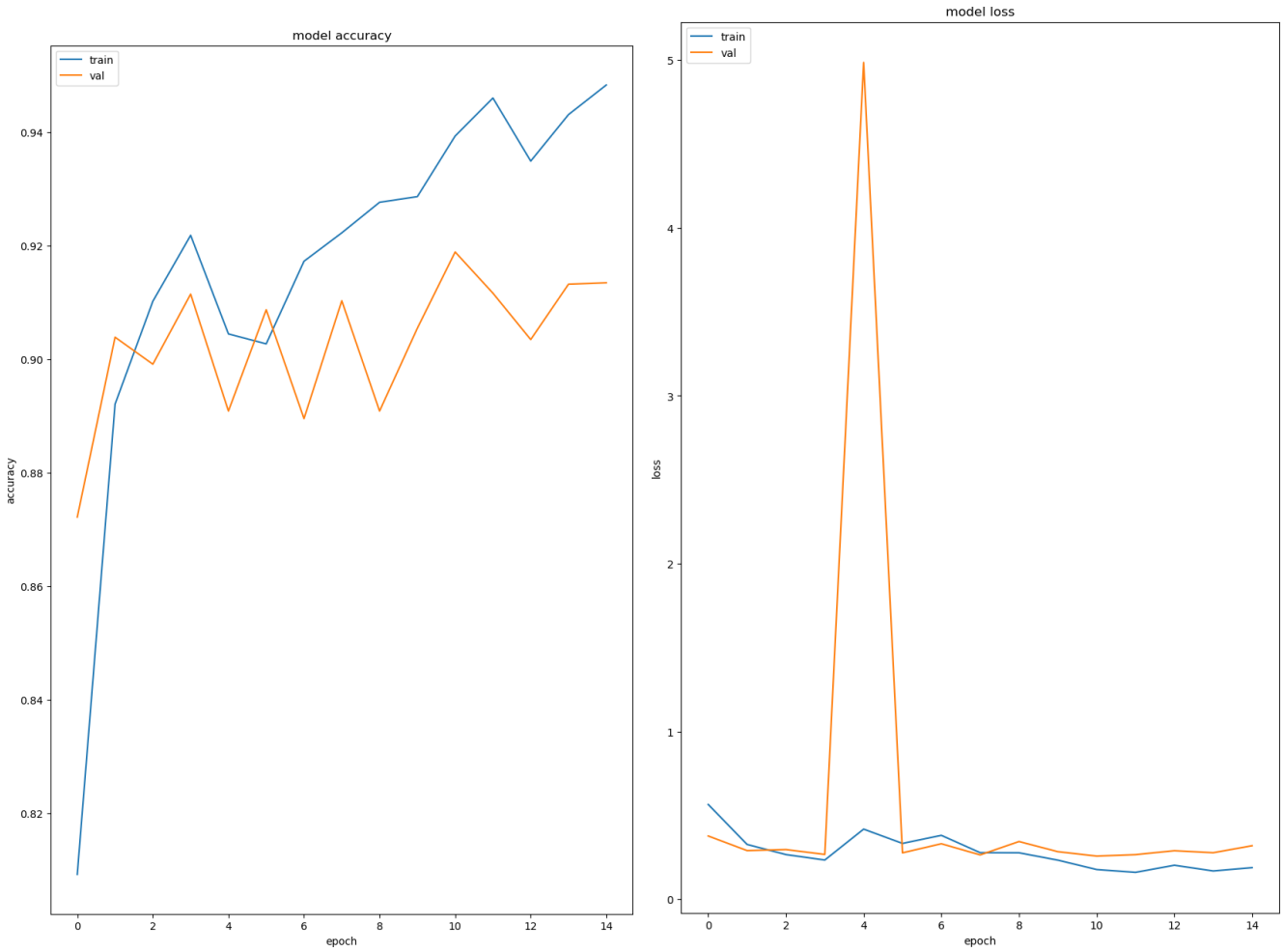


Figure 35: Evaluation Matrix of VGG-16

2. **Train ResNet152V2**:
   Same as the previous,We pass the custimazed ResNet152V2 pretrained model we got from the previous section to fit_pretrained_model. This function will fit the custimazed ResNet152V2 pretrained model and returns history plot it

   After fitting our model, we got the following curves. The Curves is pretty good. There is no overfitting.The accuracy in train data is 94.8% and accuracy in validation data is 91.3 %. So, there is no significant difference between them. .As you see there is a significant improvement in acccuracy than LeNet-5.



(a) RESNET152V2 Accuracy

(b) RESNET152V2 Loss

Figure 36: plots of RESNET152V2

we evaluated our model in test data and got 91.66% accuracy same as validation accuracy as shown in the figure below

```
accuracy_resnet , report_resnet = evaluate_model(test_X, y_test , ResNet152V2_model_)

313/313 [==============================] - 14s 34ms/step
classification report
              precision    recall  f1-score   support

           0       0.84      0.88      0.86      1000
           1       0.99      0.98      0.99      1000
           2       0.91      0.85      0.88      1000
           3       0.91      0.93      0.92      1000
           4       0.88      0.85      0.87      1000
           5       0.98      0.96      0.97      1000
           6       0.75      0.80      0.77      1000
           7       0.94      0.97      0.95      1000
           8       0.99      0.98      0.99      1000
           9       0.98      0.96      0.97      1000

    accuracy                           0.92     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.92      0.92      0.92     10000


test accuracy 0.9166
```

Figure 37: Evaluation Matrix of RESNET152V2

10 Comparison

In this section ,we will compare between models. The below table contains the accuracy of each model. From the below table, we can notice that :

1. VGG-16 is the best model It has the highest accuracy in test data.
2. VGG-16 and ResNet152V2 are better than leNet-5

Models	Test Accuracy
LeNet-5 without Cross Validation	85.44 %
LeNet-5 with Cross Validation	88.28 %
VGG-16	92.37 %
ResNet152V2	91.66 %

Figure 38: Table of models and their accuracy

11 Conclusion

I think pretrained models (VGG-16 and ResNet152V2) further improve the accuracy compared to LeNet-5.

LeNet-5 is a convolutional neural network architecture that was proposed in 1998 by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. On the other hand, pretrained models are neural networks that have already been trained on large datasets, such as ImageNet, and can be fine-tuned on a smaller and specific dataset, such as Fashion-MNIST.

In terms of performance on the Fashion-MNIST dataset, pretrained models have shown better results compared to LeNet-5. This is because pretrained models have learned a wide range of visual features from a large dataset, which can be useful in recognizing patterns and shapes in the Fashion-MNIST dataset. In contrast, LeNet-5 is a relatively simple architecture with limited capacity and may not be able to extract complex features from the images in the Fashion-MNIST dataset.

Furthermore, pretrained models can be easily fine-tuned to the Fashion-MNIST dataset, allowing for faster training and better performance compared to training a model from scratch. This is because the pretrained model has already learned basic features that are relevant to the new dataset, and the fine-tuning process can focus on optimizing the model for the specific task.

Therefore, while LeNet-5 may have been a groundbreaking architecture at the time of its proposal, pretrained models have since surpassed its performance on the Fashion-MNIST dataset.